Algorithms on Indeterminate Strings

J. Holub[1,2⋆] and W. F. Smyth[1,3⋆⋆]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
[2] Department of Computer Science & Engineering, Czech Technical University,
Karlovo náměstí 13, Praha 2, CZ-121 35, Czech Republic
[3] Department of Computing, Curtin University of Technology
GPO Box U1987, Perth WA 6845, Australia

Abstract. A recent paper describes an average-case linear-time algorithm for the calculation of all the borders of every prefix of a string $\boldsymbol{x}$, where $\boldsymbol{x}$ can contain "don't-care" letters that match with any letter of the alphabet. In this paper we show that in fact two distinct cases arise for the border calculation — we call them "quantum" and "deterministic", and both of them can arise in practice. The existing algorithm applies to the quantum case, but we show how to modify it to deal with the deterministic case. We then extend the border algorithms to apply also to what we call "indeterminate strings". We also discuss other algorithms on these indeterminate strings, some of them left as open problems.

1 Introduction

A ***border*** $\boldsymbol{u}$ of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a proper prefix of $\boldsymbol{x}$ that is also a suffix of $\boldsymbol{x}$; thus $\boldsymbol{u} = \boldsymbol{x}[1..b] = \boldsymbol{x}[n-b+1..n]$ for some $b \in 0..n-1$. The ***border array*** of $\boldsymbol{x}$ is a string $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ such that for every $i \in 1..n$, $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. In fact, since every border of any border of $\boldsymbol{x}$ is also a border of $\boldsymbol{x}$, $\boldsymbol{\beta}$ in fact encodes *all* the borders of every prefix of $\boldsymbol{x}$. The border array is computed in time $\Theta(n)$ by the so-called "failure function" algorithm [1]; since it identifies all the periods of every prefix of $\boldsymbol{x}$, the border array (or some variant of it) is an essential ingredient of many string algorithms.

Recently [5] addressed the problem of computing the border array of a string that may include occurrences of a special ***don't-care*** letter $*$, defined as follows:

⋆ Supported by an NSERC/NATO Science Fellowship and GAČR Grant No. GP201/01/P082.
⋆⋆ Supported by NSERC research grant No. OPG0008180.

Given an alphabet Σ, let $\Sigma' = \Sigma \cup \{*\}$. Then introduce a ***matching relation*** $\approx$ such that

$$\begin{array}{r} \forall \lambda, \mu \in \Sigma, \quad \mu \approx \lambda \;\Leftrightarrow\; \mu = \lambda; \\ \forall \lambda \in \Sigma', \quad * \approx \lambda \;\wedge\; \lambda \approx *. \end{array} \tag{1}$$

Thus, informally, $*$ matches every letter in Σ as well as itself. Note that the relation $\approx$ is not transitive (that is, $\lambda \approx * \approx \mu \not\Rightarrow \lambda \approx \mu$). Nevertheless, $\approx$ extends naturally to strings:

$$\boldsymbol{u}[1..n] \approx \boldsymbol{v}[1..n] \;\Leftrightarrow\; \forall i \in 1..n,\; \boldsymbol{u}[i] \approx \boldsymbol{v}[i].$$

Further, the definition of the border of a string containing don't-care letters (and so based on $\approx$) is also a natural extension of the usual border; the use of the term "quantum" is explained in the next section.

Definition 1 (Quantum Border).
A ***quantum border*** *of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ' is a proper prefix $\boldsymbol{x}[1..b]$ such that $\boldsymbol{x}[1..b] \approx \boldsymbol{x}[n-b+1..n]$.*

In [5] an algorithm is described that, according to this definition, computes all the quantum borders of every prefix of a given string on Σ'. Earlier, in [12], pattern-matching algorithms were described with patterns containing don't-care letters or patterns that were regular expressions; development and implementation of this research resulted in the `agrep` utility [11].

In this paper we seek to extend the work of [5] in three ways:

(1) In Section 2 we show that in some contexts the definition of quantum border may be inappropriate; we propose an alternate definition and describe a corresponding border algorithm for it.
(2) In Section 3 we define a class of strings ("indeterminate strings") that includes the class of strings on Σ' as a special case; we show that the border algorithms discussed here and in [5] can readily be extended to these strings.
(3) In Section 4 we discuss other algorithms on indeterminate strings, including applications of the border algorithms, leaving some of them as open problems.

2 The Deterministic Border

2.1 Discussion

We begin with an example:

Example 1. Let $\boldsymbol{x} = a{*}{*}c$. The longest border of $\boldsymbol{x}$ according to Definition 1 is 3 since $a{*}{*} \approx {*}{*}c$:

$$\begin{array}{rl} & \begin{smallmatrix}1 & 2 & 3 & 4\end{smallmatrix} \\ \boldsymbol{x} = & a * * \; c \\ & * * \; c \\ & \;\; a * * \end{array} \tag{2}$$

However, if we regard every occurrence of $*$ as a symbol that represents a single deterministic letter of Σ, is there still a border of length 3? Observe that in order to achieve this, we must have

$$\boldsymbol{x}[2] \approx \boldsymbol{x}[1] = a, \quad \boldsymbol{x}[2] \approx \boldsymbol{x}[3] = c;$$

in other words, $\boldsymbol{x}[2] = *$ must be a "quantum" variable that simultaneously matches two distinct letters!

Of course this interpretation may be legitimate. A string identified by a multiple alignment process may contain gaps that represent no specific letter. For example, to align a string $\boldsymbol{x} = ac$ with two others, $\boldsymbol{y} = agtc$ and $\boldsymbol{z} = atgc$, we might introduce gaps into $\boldsymbol{x}$, yielding

$$\boldsymbol{x'} = a\_\_c.$$

These gaps merely represent a means of satisfying an algorithm, they have in fact already matched both g and t during the alignment process; therefore it makes sense to treat them as quantum variables.

On the other hand, it may be that each don't-care letter is a placeholder for a letter of Σ that has yet to be specified. We thus have the freedom to replace any occurrence of $*$ by any letter of Σ, but only by a single letter: a single occurrence of $*$ cannot match two different letters of Σ at the same time. In this case, then, $*$ is a "deterministic" variable.

Given any string $\boldsymbol{x}$ on Σ', we say that $\widehat{\boldsymbol{x}}$ on Σ is an ***assignment*** of $\boldsymbol{x}$ if and only if $\widehat{\boldsymbol{x}} \approx \boldsymbol{x}$. We have then

Definition 2 (Deterministic Border).
*A **deterministic border** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ' is a proper prefix $\boldsymbol{x}[1..b]$ such that, for some assignment $\widehat{\boldsymbol{x}}$ of $\boldsymbol{x}$, $\widehat{\boldsymbol{x}}[1..b] = \widehat{\boldsymbol{x}}[n-b+1..n]$.*

We remark that for $b \leq n/2$, $\boldsymbol{u} = \boldsymbol{x}[1..b]$ is a deterministic border if and only if it is also a quantum border. Further, for strings $\boldsymbol{x}$ on Σ' that do not in fact contain $*$, observe that both the quantum and deterministic borders reduce to the usual border.

Example 2 (Continuing Example 1). The longest deterministic border of $\boldsymbol{x} = a{*}{*}c$ has length 2:

$$\begin{array}{rcccc} & 1 & 2 & 3 & 4 \\ \boldsymbol{x} = & a & * & * & c \\ & * & c & & \\ & & & a & * \end{array} \tag{3}$$

2.2 The Algorithm

We display Algorithm 1, computation of the usual border array [1], as a point of departure for our new algorithms. Recall that all $\kappa \geq 1$ borders of each prefix $\boldsymbol{x}[1..i]$, $1 \leq i \leq n$, are given in descending order of length by

$$\boldsymbol{\beta^1}[i] = \boldsymbol{\beta}[i], \quad \boldsymbol{\beta^j}[i] = \boldsymbol{\beta}\left[\boldsymbol{\beta^{j-1}}[i]\right], \; j = 2, 3, \ldots, \kappa, \tag{4}$$

where κ is the least positive integer such that $\boldsymbol{\beta^\kappa}[i] = 0$. Thus $\boldsymbol{\beta}[i+1]$ either equals $\boldsymbol{\beta^j}[i] + 1$, where j is the least integer such that $\boldsymbol{x}[i+1] = \boldsymbol{x}\left[\boldsymbol{\beta^j}[i] + 1\right]$, or else, if no such j exists, equals zero. This is the processing carried out for every i by Algorithm 1: for $b = \boldsymbol{\beta^j}[i]$, the **while** loop identifies the smallest j (largest b) such that $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$. It then follows that every border of $\boldsymbol{x}[b+1]$ is also a border of $\boldsymbol{x}[i+1]$.

Algorithm 1 Computing the border array for $\boldsymbol{x} \in \Sigma^*$

$\boldsymbol{\beta}[1] \leftarrow 0$
for $i \leftarrow 1$ to $n-1$ **do**
  $b \leftarrow \boldsymbol{\beta}[i]$
  while $b > 0$ **and** $\boldsymbol{x}[i+1] \neq \boldsymbol{x}[b+1]$ **do**
    $b \leftarrow \boldsymbol{\beta}[b]$
  end while
  if $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$ **then**
    $\boldsymbol{\beta}[i+1] \leftarrow b+1$
  else
    $\boldsymbol{\beta}[i+1] \leftarrow 0$
  end if
end for

However, by virtue of the nontransitivity of $\approx$, this convenient relation does not hold for quantum and deterministic borders. For example, given $\boldsymbol{x} = a{*}c$, $\boldsymbol{x}[1..2]$ has borders of length 1 and 0, while $\boldsymbol{x}[1..3]$ has a border of length 2, but not of length 1. Thus, as shown in Algorithm 2 [5], it becomes necessary to store a list β_i of all borders corresponding to each prefix $\boldsymbol{x}[1..i]$. This list is then used instead of (4) to compute the borders of $\boldsymbol{x}[1..i+1]$, and the collection of lists β_i, $i = 1, 2, \ldots, n$, becomes the output of the algorithm rather than a single border array $\boldsymbol{\beta}$. Alternatively, if only the longest border at each position is required, we can output a border array formed from the first (maximum) value in each of the n lists β_i.

Algorithm 2 Computing the quantum borders

$\beta_i \leftarrow \emptyset, \quad \forall i \in \{1..n\}$
for $i \leftarrow 1$ to $n-1$ **do**
  for all $b \in \beta_i$ **do**
    if $\boldsymbol{x}[b+1] \approx \boldsymbol{x}[i+1]$ **then**
      $\beta_{i+1} \leftarrow \beta_{i+1} \cup \{b+1\}$
    end if
  end for
  if $\boldsymbol{x}[i+1] \approx \boldsymbol{x}[1]$ **then** {match with $\boldsymbol{x}[1]$}
    $\beta_{i+1} \leftarrow \beta_{i+1} \cup \{1\}$
  end if
end for

For deterministic borders, the calculation becomes somewhat more complicated. As shown in Table 1, the list entries at each position need to be pairs (b, λ) that give not only the length b of each border but also the assignment λ that corresponds to b. A pair $(b, \lambda) \in \beta_i$ tells us that $\boldsymbol{x}[1..b] \approx \boldsymbol{x}[i-b+1..i]$ corresponding to the assignment $\boldsymbol{x}[i] = \lambda \approx \boldsymbol{x}[b]$.

Our objective is to construct the list β_i by an inspection of each element $(b, \lambda) \in \beta_i$. If $b \leq i/2$, then as noted above we need only perform the same processing as in the quantum case, since no conflict in assignment can arise. However, for $b > i/2$, the prefix $\boldsymbol{x}[1..i]$ takes the form $\boldsymbol{u}^r\boldsymbol{u'}$, where $\boldsymbol{u} = \boldsymbol{x}[1..i-b]$, $r \geq 2$, and $\boldsymbol{u'}$ is a proper prefix of $\boldsymbol{u}$. Thus $i-b$ is a period of $\boldsymbol{x}[1..i]$. In order that $i-b$ should also be a period of $\boldsymbol{x}[1..i+1]$, we must have $\boldsymbol{x}[i+1]$ matching an assignment of $\boldsymbol{x}[b+1]$ that corresponds to period $i-b$. This assignment is given by the element

$$\big((b+1)-(i-b), \mu\big) = (2b-i+1, \mu)$$

in the list β_{b+1}; if this element exists and $\mu \approx \boldsymbol{x}[i+1]$, then $\boldsymbol{x}[1..i+1]$ has deterministic border $b+1$, and the element $\big(b+1, \boldsymbol{x}[i+1]\big)$ is added to list β_{i+1}.

As it turns out, the processing required corresponding to each element $(b, \lambda) \in \beta_i$ breaks down into five cases, described below. In order to describe the processing simply, it is convenient to represent the lists β_i as arrays $\boldsymbol{\beta_i}[1..n]$, defined as follows: if (b, λ) is an element of the list β_i, then $\boldsymbol{\beta_i}[b] = \lambda$; if there is no element (b, λ) in β_i, then $\boldsymbol{\beta_i}[b] = \emptyset$. For ease of discussion, we describe the processing in terms of the arrays $\boldsymbol{\beta_i}$, even though for efficiency it has been implemented in terms of lists.

Algorithm 3 Computing the deterministic borders

1: $\beta_i[j] \leftarrow \emptyset, \quad \forall i, j \in \{1..n\}$
2: **for** $i \leftarrow 1$ to $n-1$ **do**
3:   **for all** $b, \beta_i[b] \neq \emptyset$ **do**
4:     **if** $\boldsymbol{x}[b+1] = \boldsymbol{x}[i+1] \neq *$ **then** {case 1}
5:       $\beta_{i+1}[b+1] \leftarrow \boldsymbol{x}[i+1]$
6:     **else if** $\boldsymbol{x}[b+1] = \boldsymbol{x}[i+1] = *$ **then** {case 3}
7:       **if** $2b - i + 1 < 1$ **then**
8:         $\beta_{i+1}[b+1] \leftarrow *$
9:       **else**
10:         $\beta_{i+1}[b+1] \leftarrow \boldsymbol{x}[\beta_{b+1}[2b - i + 1]]$
11:       **end if**
12:     **else if** $\boldsymbol{x}[b+1] \neq *$ **and** $\boldsymbol{x}[i+1] = *$ **then** {case 4}
13:       $\beta_{i+1}[b+1] \leftarrow \boldsymbol{x}[b+1]$
14:     **else if** $\boldsymbol{x}[b+1] = *$ **and** $\boldsymbol{x}[i+1] \neq *$ **then** {case 5}
15:       **if** $2b - i + 1 < 1$ **or** $\boldsymbol{x}[\beta_{b+1}[2b - i + 1]] \approx \boldsymbol{x}[i+1]$ **then**
16:         $\beta_{i+1}[b+1] \leftarrow \boldsymbol{x}[i+1]$
17:       **end if**
18:     **end if**
19:   **end for** {Next deal with empty border at i}
20:   **if** $\boldsymbol{x}[i+1] = \boldsymbol{x}[1]$ **or** $\boldsymbol{x}[i+1] = *$ **then** {match with $\boldsymbol{x}[1]$}
21:     $\beta_{i+1}[1] \leftarrow \boldsymbol{x}[1]$
22:   **else if** $\boldsymbol{x}[1] = *$ **then**
23:     $\beta_{i+1}[1] \leftarrow \boldsymbol{x}[i+1]$
24:   **end if**
25: **end for**

The five cases are as follows (lines 4–18 in Algorithm 3):

1. $\boldsymbol{x}[b+1] = \boldsymbol{x}[i+1] \neq *$ (two matching symbols of Σ): The border can be extended (i.e., $\boldsymbol{x}[1..i+1]$ has a border of length $b+1$).
2. $\boldsymbol{x}[b+1] \neq \boldsymbol{x}[i+1] \wedge \boldsymbol{x}[b+1] \neq * \neq \boldsymbol{x}[i+1]$ (two mismatching symbols of Σ): The border b cannot be extended.
3. $\boldsymbol{x}[b+1] = \boldsymbol{x}[i+1] = *$: The border can be extended. For $2b-i+1 \geq 1$, we only have to copy the assignment requirement for $\boldsymbol{x}[b+1]$ to $\boldsymbol{x}[i+1]$; that is, $\beta_{i+1}[b+1] \leftarrow \beta_{b+1}[2b-i+1]$. For $2b-i+1 < 1$, $\boldsymbol{x}[b+1]$ has no assignment requirement since $\boldsymbol{x}[b+1]$ has not been compared with any other symbol; $b+1$ is less than the period of the border.
4. $\boldsymbol{x}[b+1] \neq * \;\wedge\; \boldsymbol{x}[i+1] = *$: The border can be extended; we just have to $\beta_{i+1}[b+1] \leftarrow \boldsymbol{x}[b+1]$.

5. $\boldsymbol{x}[b+1] = * \wedge \boldsymbol{x}[i+1] \neq *$: The border can be extended only if there is no conflict between $\boldsymbol{x}[i+1]$ and the assignment requirement for $\boldsymbol{x}[b+1]$. If $\beta_{b+1}[2b-i+1] = \boldsymbol{x}[i+1]$ or if $\boldsymbol{x}[b+1]$ has no assignment requirement ($\beta_{b+1}[2b-i+1] = *$ or $2b-i+1 < 1$), we can extend the border.

In case 5 it is not necessary to store the required assignment for β_{i+1} back to β_{b+1}, since it will not be tested in the following steps. This is a consequence of

Note 1. Each pair $(b', \mu) \in \beta_{b+1}$, $1 \leq b < n-1$, is tested at most once: precisely when trying to extend a border b to $b+1$ at position $i+1$, where $i - b = (b+1) - b'$ is the period of the border.

A further consequence of this remark is that when we do not need to know all the required assignments at the end of computation, we do not need to store all lists β_i, $1 < i \leq n$. At each step i of the computation we need to store only π_b pairs for each border b, where π_b is the period determined by b.

The worst-case time and space complexity of Algorithm 3 is $\mathcal{O}(n^2)$ but the expected time and space complexity is $\mathcal{O}(n)$, since the expected number of borders of any string is less than 3.5 (see [5]).

The following example illustrates the execution of the new algorithm:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	a	a	b	a	$*$	a	$*$	a	b	$*$	$*$
β		$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
			$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
							$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
								$(1,a)$	$(2,a)$			
									$(1,a)$	$(2,a)$		
										$(1,a)$		
											$(1,a)$	$(2,a)$
												$(1,a)$

Table 1. Lists β_i of pairs for string $\boldsymbol{x} = aaaba{*}a{*}ab{*}{*}$.

3 Indeterminate Strings

In this section we introduce a class of strings that have a more general don't-care capability than the strings on Σ'. For any nonempty subset

$S \subset \Sigma$, let

$$\Psi(S) \tag{5}$$

denote any one of the letters of S. Thus for any $\lambda \in \Sigma$, we write

$$\Psi(\lambda) = \Psi\big(\{\lambda\}\big) = \lambda,$$

while at the other extreme, $\Psi(\Sigma) = *$. Thus we introduce all possible restricted don't-care letters, in addition to the unrestricted one denoted by $*$. Let Σ'' denote the alphabet whose elements are given by (5), where S ranges over all nonempty subsets of Σ. Thus Σ'' is the power set of Σ less the empty set, and so

$$|\Sigma''| = 2^{|\Sigma|} - 1.$$

For brevity we represent concatenation in an abbreviated form:

$$\Psi(S_1)\Psi(S_2) = \Psi(S_1, S_2).$$

Definition 3 (Indeterminate String).
An ***indeterminate string*** *of length n on an alphabet Σ'' is a string*

$$\Psi(S_1, S_2, \ldots, S_n),$$

where each S_i, $1 \le i \le n$, is a nonempty subset of Σ.

We can define the quantum border and the deterministic border of indeterminate strings in terms directly analogous to Definitions 1 and 2.

In order to compute the quantum border of a string on Σ'', we use a modified Algorithm 2. The modification ensures constant time for computing the matching relation $\approx$ for any two letters of Σ''. To achieve this, we introduce for every symbol $\lambda \in \Sigma''$ a bit vector of length $|\Sigma|$ as follows:

$$\begin{aligned} \forall \lambda \in \Sigma'', \quad & \nu[\lambda] = [\nu_{\lambda_1}, \nu_{\lambda_2}, \ldots, \nu_{\lambda_{|\Sigma|}}], \\ & \text{where } \forall \alpha \in \Sigma, \nu_\alpha = \begin{cases} 1, & \text{if } \alpha \approx \lambda, \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{6}$$

At the beginning of the algorithm we set the bit vector ν_i for each position $i \in 1..n$ in $\boldsymbol{x}[1..n]$. Then we proceed as in Algorithm 2, but instead of comparing $\boldsymbol{x}[b+1]$ and $\boldsymbol{x}[i+1]$, we test bit vectors ν_{b+1} and ν_{i+1} using the bit operation AND. Thus we can compare any two symbols of Σ'' in constant time. The resulting algorithm is shown in Algorithm 4.

Algorithm 4 Computing quantum border array for string $\boldsymbol{x} \in \Sigma''^{*}$

$\beta_i \leftarrow \emptyset, \quad \forall i \in \{1..n\}$
$\nu_i \leftarrow \nu[\boldsymbol{x}[i]], \quad \forall i \in \{1..n\}$ $\quad$ {set bit vector for each $\boldsymbol{x}[i]$}
for $i \leftarrow 1$ to $n-1$ **do**
  for all $b \in \beta_i$ **do**
    if ν_{b+1} AND $\nu_{i+1} \neq 0^{|\Sigma|}$ **then**
      $\beta_{i+1} \leftarrow \beta_{i+1} \cup \{b+1\}$
    end if
  end for
  if ν_1 AND $\nu_{i+1} \neq 0^{|\Sigma|}$ **then** {match with $\boldsymbol{x}[1]$}
    $\beta_{i+1} \leftarrow \beta_{i+1} \cup \{1\}$
  end if
end for

In order to compute the deterministic border of a string $\boldsymbol{x} \in \Sigma''^{*}$, we extend Algorithm 4 so that each list β_i contains a list of pairs (b, ν_a), where ν_a represents the required assignment. When comparing $\boldsymbol{x}[b+1]$ and $\boldsymbol{x}[i+1]$, we also perform the bit operation AND on the vector ν_{i+1} and the assignment requirement stored in β_{b+1}. Unlike in Algorithm 4, in this case we store the nonzero result in β_{i+1} so that this value is tested next time instead of ν_{i+1} when trying to extend the same border. As in Algorithm 3, we have to test wheather $2b - i + 1 < 0$; if so, no assignment requirement has been established, so we do a direct test of ν_{b+1}.

Algorithm 5 Computing deterministic border for string $\boldsymbol{x} \in \Sigma''^{*}$

$\beta_i[j] \leftarrow \emptyset, \quad \forall i, j \in \{1..n\}$
$\nu_i \leftarrow \nu[\boldsymbol{x}[i]], \quad \forall i \in \{1..n\}$ $\quad$ {set bit vector for each $\boldsymbol{x}[i]$}
for $i \leftarrow 1$ to $n-1$ **do**
  for all $b, \beta_i[b] \neq \emptyset$ **do**
    if $2b - i + 1 < 0$ **then**
      $p \leftarrow \nu_{b+1}$ AND ν_{i+1}
    else
      $p \leftarrow \beta_i[b+1]$ AND ν_{i+1}
    end if
    if $p \neq 0^{|\Sigma|}$ **then**
      $\beta_{i+1}[b+1] \leftarrow p$
    end if
  end for
  $p \leftarrow \nu_1$ AND ν_{i+1}
  if $p \neq 0^{|\Sigma|}$ **then**
    $\beta_{i+1}[1] \leftarrow p$
  end if
end for

We remark that in both of these algorithms we need storage only for letters of Σ'' that actually occur in the string $\boldsymbol{x}$; thus even though $|\Sigma''|$ is exponential with respect to $|\Sigma|$, in practice the algorithms remain efficient.

We remark also that the indeterminate strings introduced in this section are special cases of regular expressions. We leave for future study the extension of these and related algorithms to strings that are expressed as regular expressions in their full generality.

4 Applications & Open Problems

The border array calculation and its many closely-related variants are ubiquitous among string algorithms; for example,

- the Knuth-Morris-Pratt pattern-matching algorithm [6] and its variants;
- the Boyer-Moore pattern-matching algorithm [2] and its numerous variants;
- Duval's algorithm for Lyndon word decomposition [3];
- Main & Lorentz's repetitions algorithm [9] and Main's repetitions algorithm [8];
- the calculation of the cover array [7].

The extensions of these algorithms to execute using the border array variants described in this paper remain open problems.

Further we note that there are other possible applications of indeterminate strings not related to border arrays:

- suffix tree and suffix array calculations;
- pattern-matching algorithms such as Horspool's [4] and Sunday's [10];
- algorithms using the techniques of nondeterministic and deterministic finite automata.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.
2. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
3. J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
4. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

5. C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. G. Perdikuri, W. F. Smyth, and A. K. Tsakalidis. String regularities with don't cares. In M. Balík and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference '02*, pages 65–74, Czech Technical University, Prague, Czech Republic, 2002. Research Report DC–2002–03.
6. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
7. Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
8. M. G. Main. Detecting leftmost maximal periodicities. *Discret. Appl. Math.*, 25:145–153, 1989.
9. M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
10. D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
11. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.
12. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.